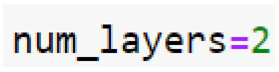
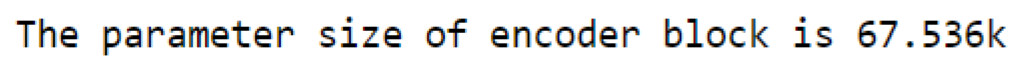
- Screenshot of task-1
  - Number of layers



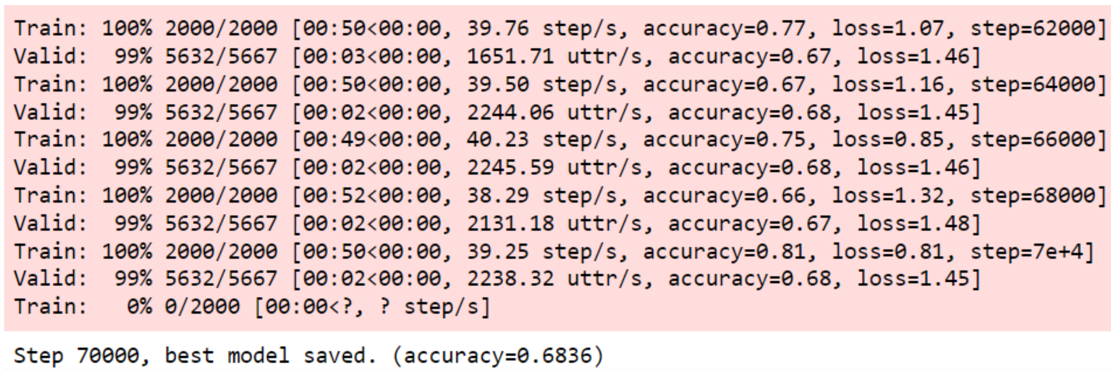
```
num_layers=2
```

  - Parameter size



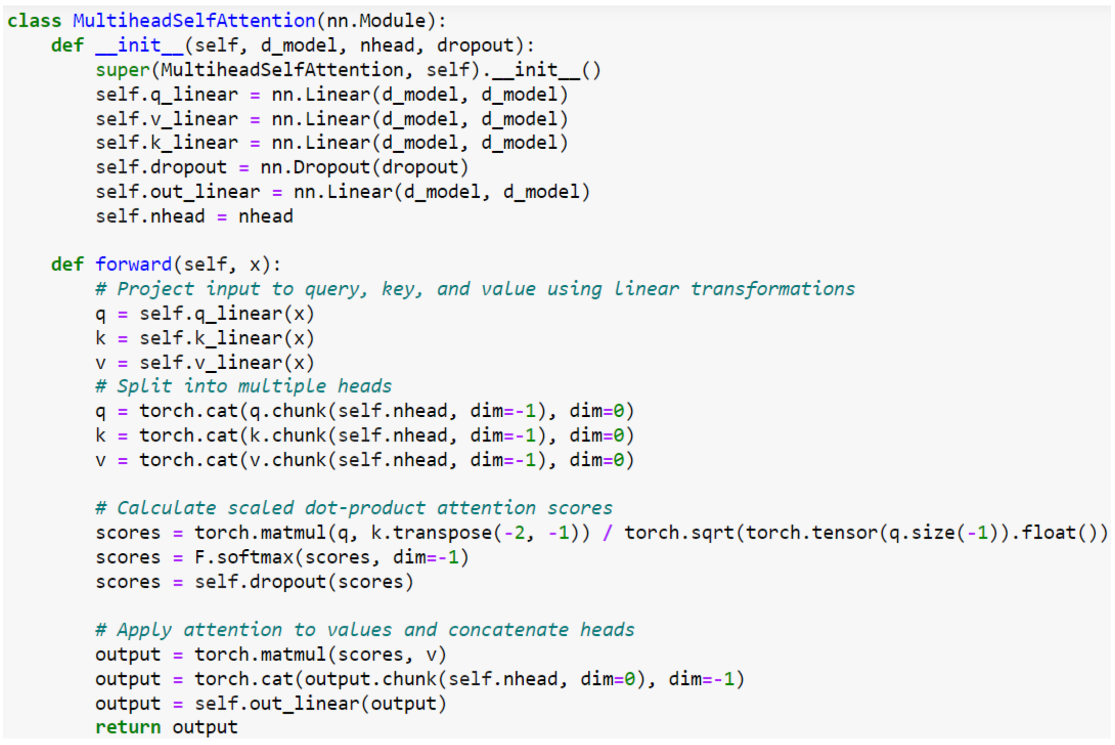
```
The parameter size of encoder block is 67.536k
```

  - Accuracy



```
Train: 100% 2000/2000 [00:50<00:00, 39.76 step/s, accuracy=0.77, loss=1.07, step=62000]
Valid:  99% 5632/5667 [00:03<00:00, 1651.71 uttr/s, accuracy=0.67, loss=1.46]
Train: 100% 2000/2000 [00:50<00:00, 39.50 step/s, accuracy=0.67, loss=1.16, step=64000]
Valid:  99% 5632/5667 [00:02<00:00, 2244.06 uttr/s, accuracy=0.68, loss=1.45]
Train: 100% 2000/2000 [00:49<00:00, 40.23 step/s, accuracy=0.75, loss=0.85, step=66000]
Valid:  99% 5632/5667 [00:02<00:00, 2245.59 uttr/s, accuracy=0.68, loss=1.46]
Train: 100% 2000/2000 [00:52<00:00, 38.29 step/s, accuracy=0.66, loss=1.32, step=68000]
Valid:  99% 5632/5667 [00:02<00:00, 2131.18 uttr/s, accuracy=0.67, loss=1.48]
Train: 100% 2000/2000 [00:50<00:00, 39.25 step/s, accuracy=0.81, loss=0.81, step=7e+4]
Valid:  99% 5632/5667 [00:02<00:00, 2238.32 uttr/s, accuracy=0.68, loss=1.45]
Train:   0% 0/2000 [00:00<?, ? step/s]

Step 70000, best model saved. (accuracy=0.6836)
```

  - MultiheadSelfAttention



```python
class MultiheadSelfAttention(nn.Module):
    def __init__(self, d_model, nhead, dropout):
        super(MultiheadSelfAttention, self).__init__()
        self.q_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)
        self.out_linear = nn.Linear(d_model, d_model)
        self.nhead = nhead

    def forward(self, x):
        # Project input to query, key, and value using linear transformations
        q = self.q_linear(x)
        k = self.k_linear(x)
        v = self.v_linear(x)
        # Split into multiple heads
        q = torch.cat(q.chunk(self.nhead, dim=-1), dim=0)
        k = torch.cat(k.chunk(self.nhead, dim=-1), dim=0)
        v = torch.cat(v.chunk(self.nhead, dim=-1), dim=0)

        # Calculate scaled dot-product attention scores
        scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(q.size(-1)).float())
        scores = F.softmax(scores, dim=-1)
        scores = self.dropout(scores)

        # Apply attention to values and concatenate heads
        output = torch.matmul(scores, v)
        output = torch.cat(output.chunk(self.nhead, dim=0), dim=-1)
        output = self.out_linear(output)
        return output
```

- FeedForward

```python
class FeedForward(nn.Module):
    def __init__(self, d_model, dim_feedforward, dropout):
        super(FeedForward, self).__init__()
        # Linear layer for the first feedforward network
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        # Dropout for regularization
        self.dropout = nn.Dropout(dropout)
        # Linear layer for the second feedforward network
        self.linear2 = nn.Linear(dim_feedforward, d_model)

    def forward(self, x):
        # Apply ReLU activation to the first linear layer
        x = F.relu(self.linear1(x))
        # Apply dropout for regularization
        x = self.dropout(x)
        # Apply the second linear layer
        x = self.linear2(x)
        return x
```

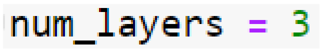
- TransformerEncoderLayer

```python
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, dim_feedforward, nhead, dropout):
        super(TransformerEncoderLayer, self).__init__()
        # Multi-head self-attention layer
        self.self_attn = MultiheadSelfAttention(d_model, nhead, dropout)
        # Feedforward network
        self.feed_forward = FeedForward(d_model, dim_feedforward, dropout)
        # Layer normalization for the self-attention output
        self.norm1 = nn.LayerNorm(d_model)
        # Layer normalization for the feedforward output
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # Self-attention
        attn_output = self.self_attn(x)
        x = x + self.dropout(attn_output)
        x = self.norm1(x)

        # Feedforward
        ff_output = self.feed_forward(x)
        x = x + self.dropout(ff_output)
        x = self.norm2(x)

        return x
```

- TransformerEncoder

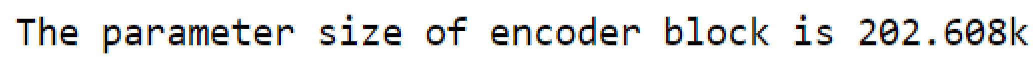
```
class TransformerEncoder(nn.Module):
    def __init__(self, encoder_layer, num_layers):
        super(TransformerEncoder, self).__init__()
        # Stack multiple transformer encoder layers
        self.layers = nn.ModuleList([encoder_layer for _ in range(num_layers)])

    def forward(self, x):
        # Pass input through all transformer encoder layers
        for layer in self.layers:
            x = layer(x)
        return x
```

- Screenshot of task-2
  - Number of layers



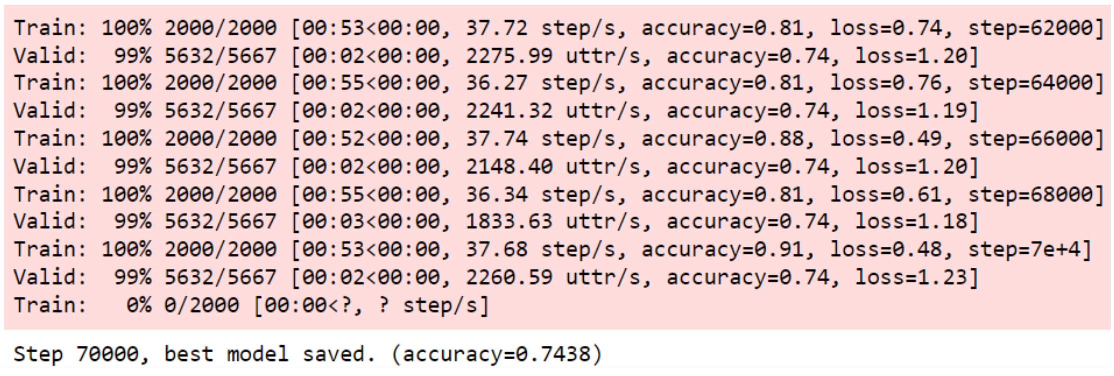
```
num_layers = 3
```

  - Parameter size



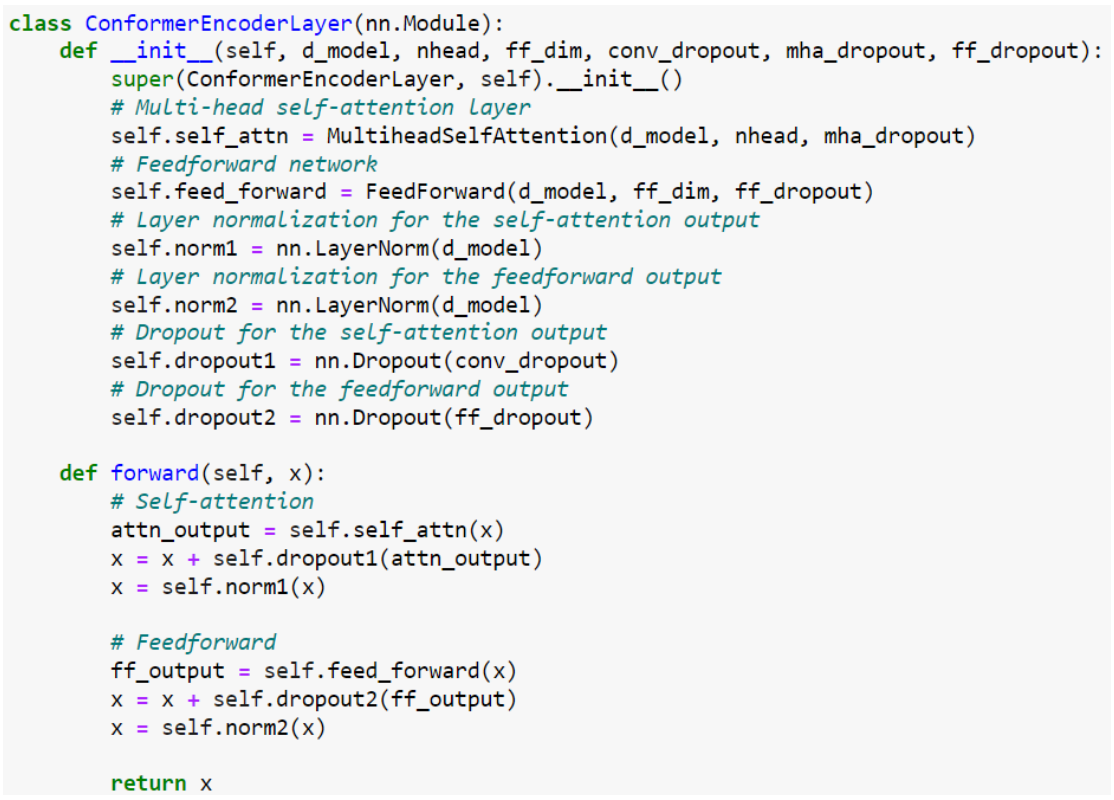
```
The parameter size of encoder block is 202.608k
```

  - Accuracy



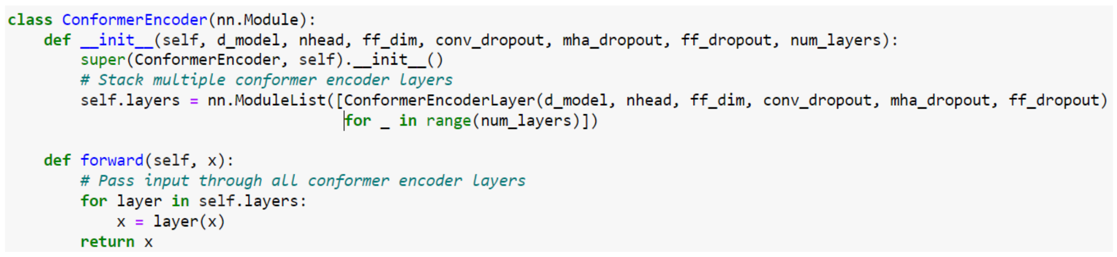
```
Train: 100% 2000/2000 [00:53<00:00, 37.72 step/s, accuracy=0.81, loss=0.74, step=62000]
Valid:  99% 5632/5667 [00:02<00:00, 2275.99 uttr/s, accuracy=0.74, loss=1.20]
Train: 100% 2000/2000 [00:55<00:00, 36.27 step/s, accuracy=0.81, loss=0.76, step=64000]
Valid:  99% 5632/5667 [00:02<00:00, 2241.32 uttr/s, accuracy=0.74, loss=1.19]
Train: 100% 2000/2000 [00:52<00:00, 37.74 step/s, accuracy=0.88, loss=0.49, step=66000]
Valid:  99% 5632/5667 [00:02<00:00, 2148.40 uttr/s, accuracy=0.74, loss=1.20]
Train: 100% 2000/2000 [00:55<00:00, 36.34 step/s, accuracy=0.81, loss=0.61, step=68000]
Valid:  99% 5632/5667 [00:03<00:00, 1833.63 uttr/s, accuracy=0.74, loss=1.18]
Train: 100% 2000/2000 [00:53<00:00, 37.68 step/s, accuracy=0.91, loss=0.48, step=7e+4]
Valid:  99% 5632/5667 [00:02<00:00, 2260.59 uttr/s, accuracy=0.74, loss=1.23]
Train:   0% 0/2000 [00:00<?, ? step/s]

Step 70000, best model saved. (accuracy=0.7438)
```

  - ConformerEncoderLayer



```python
class ConformerEncoderLayer(nn.Module):
    def __init__(self, d_model, nhead, ff_dim, conv_dropout, mha_dropout, ff_dropout):
        super(ConformerEncoderLayer, self).__init__()
        # Multi-head self-attention layer
        self.self_attn = MultiheadSelfAttention(d_model, nhead, mha_dropout)
        # Feedforward network
        self.feed_forward = FeedForward(d_model, ff_dim, ff_dropout)
        # Layer normalization for the self-attention output
        self.norm1 = nn.LayerNorm(d_model)
        # Layer normalization for the feedforward output
        self.norm2 = nn.LayerNorm(d_model)
        # Dropout for the self-attention output
        self.dropout1 = nn.Dropout(conv_dropout)
        # Dropout for the feedforward output
        self.dropout2 = nn.Dropout(ff_dropout)

    def forward(self, x):
        # Self-attention
        attn_output = self.self_attn(x)
        x = x + self.dropout1(attn_output)
        x = self.norm1(x)

        # Feedforward
        ff_output = self.feed_forward(x)
        x = x + self.dropout2(ff_output)
        x = self.norm2(x)

        return x
```

- ConformerEncoder



```
class ConformerEncoder(nn.Module):
    def __init__(self, d_model, nhead, ff_dim, conv_dropout, mha_dropout, ff_dropout, num_layers):
        super(ConformerEncoder, self).__init__()
        # Stack multiple conformer encoder layers
        self.layers = nn.ModuleList([ConformerEncoderLayer(d_model, nhead, ff_dim, conv_dropout, mha_dropout, ff_dropout)
                                     for _ in range(num_layers)])

    def forward(self, x):
        # Pass input through all conformer encoder layers
        for layer in self.layers:
            x = layer(x)
        return x
```

- In task-2
  - <u>Which kind of transformer-like model do you choose？</u>

    Conformer

  - <u>The reason why you choose this model.</u>
    1. Conformer 主要設計目的是優化音頻信號的處理，尤其是語音識別
    2. 使用 Relative Positional Encoding 處理音頻信號的時間序列特性

  - <u>The advantage of chosen model.</u>
    1. 針對大範圍、前後有關聯的特徵的提取能力較 transformer 佳
    2. 針對局部特徵的提取能力較 transformer 佳
    3. 引入 Convolution Block 能夠更好的捕捉音頻信號中的局部結構
    4. 引入 Depthwise Separable Convolution 可以減少參數數量，提高執行效率
    5. 引入一些針對音頻 data 的特殊結構和機制，可以更好的捕捉聲音的時間性質